Logic-Based Program Synthesis: State-of-the-Art and Future Trends

Steve Roach
Department of Computer Science
University of Texas at El Paso
El Paso, Texas 79968
sroach@cs.utep.edu

Abstract

Constructing certifiably reliable software systems is difficult. Deductive program synthesis techniques (Flener 1995, Manna and Waldinger 1980) can currently be used to construct small software systems or to organize small sets of software components in a reliable manner. In order for synthesis techniques to be applicable to real-world problems outside the experimental laboratory, they must be inexpensive relative to manual techniques. The difficulty and expense in constructing software synthesis systems currently precludes the use of these techniques in many instances.

Amphion and Meta-Amphion

Amphion (Stickel, et al. 1994) is a deductive synthesis system that has been used to construct programs in the domains of celestial mechanics and avionics. The experiences gained in the Amphion system mirror experiences in other synthesis systems. Amphion is a domain-independent system that is tailored to a domain in part through the creation of a declarative domain theory. Problem specifications are solved by programs constructed of sequences of calls to software components. Program construction is entirely automated. Programs have been generated that are currently in use by space scientists planning observations for the Cassini mission to Saturn (Roach and Van Baalen 1996, Roach, Lowry, and Pressburger 1995).

An Amphion domain theory is written in first-order logic and relates abstract, specification-level functions and predicates to concrete, implementation-level components. Specifications for programs are also written in first-order logic and take the form *Forall (inputs) Exists (outputs) ({properties})*. A general-purpose resolution theorem prover finds ground instances of the existential variables for which the set of properties hold. These ground instances form functional terms that are translated into a target language compatible with the existing software components.

While it is not particularly difficult to create a declarative domain theory for Amphion that captures the relationships between the abstract and the concrete, the performance of the general-purpose resolution theorem prover quickly degrades due to the exponential behavior of the required search. Thus, a naive domain theory can only be used to construct simple programs. In order to synthesize non-trivial programs, it is necessary to tune the domain theory. Tuning a domain theory consists of rewriting axioms to take advantage of knowledge of the implementation of the theorem prover or incorporating specialized inference mechanisms (such as decision procedures) that are tied directly to the theorem prover. Both of these methods require a high degree of expertise, a great deal of time, and are quite difficult. While the construction of decision procedures can be automated to some extent (Van Baalen and Roach 1998, Roach, Van Baalen, and Lowry 1997, Roach 97), the integration of these procedures with the general-purpose theorem prover used in Amphion has been difficult and un-maintainable.

Difficulties in Program Synthesis

In the past thirty years, a great deal of progress has been made in the development of program synthesis systems based on theorem proving, transformations, and logic programming. However, in spite of this progress, these techniques are not in the mainstream of software development. Formal program synthesis techniques, at least with the current synthesis technologies, are not appropriate for all software development situations. The characteristics of inappropriate situations include having little potential for reuse (to amortize the cost of constructing the synthesis system) and having a domain or class of problems that are not well understood.

In situations where it is necessary to prototype a system in order to answer fundamental questions about the capability of an approach or to explore domain knowledge, it is much more difficult to construct a synthesis system than to construct programs by hand. Many market-driven software systems fall into this category. Such systems are inherently difficult to formalize. While some argue that the lack of formalization is a deficiency on the part of program developers, it is frequently a necessity. It may be that the cost of formalizing a specification is too high relative to the cost of having a human interpret an informal specification. The translation between informal and formal (a task we assume to require human oversight) may be faster at lower levels of abstraction for some problems. This occurs when relatively simple ideas expressed informally become difficult to formalize.

Additionally, many synthesis techniques scale badly. Deductive techniques have exponential behavior. Thus, while they may work reasonably well for small problems, they do not work for large problems. There are approaches to addressing this problem (Roach 1997, Srinivas and McDonald 1996, Smith 1991); however, it is still difficult to reuse the work done in one domain to solve problems in another domain.

The future of program synthesis

By looking at the successes in program synthesis, it is reasonable to suggest characteristics of situations where synthesis is appropriate. In order to become a mainstream technique, synthesis must be advantageous either by making the software faster to produce, cheaper to produce and maintain, or of higher quality. The mechanisms for achieving this include

a) producing code faster via synthesis than by hand by automating tedious details of development;
b) producing code of higher quality or of higher certification than hand-development;
c) reducing the level of expertise required for practitioners to construct software.

The properties of systems amenable to economic application of synthesis fall into two categories: the simple and the complex. With simple systems, synthesis relieves programmers of tedious and repetitive programming tasks. Just as compilers relieved programmers of the task of allocating and managing storage, synthesis systems can alleviate the cumbersome tasks of managing tedious tasks. One of the advantages of Amphion's synthesis system is that a simple algorithm is implemented in a syntactically correct form. One approach to using Amphion is to create a program that solves part of a problem, then hand-modify the resulting code to complete the system. The tedious work of variable declarations, type checking, and matching parameters and arguments when combining components is handled by Amphion. The less-easily specified parts of the system (such as "display the date and time in a readable font out of the way of interesting parts of the scene") are coded by hand.

Humans have difficulty formulating plans in complex systems where it is necessary to account for a large number of interactions (Dorner 1996). It may be theoretically possible to predict the effect of some action on a system, but the large number of competing issues prevents humans from choosing an appropriate action. In software development, these situations may arise from the interactions of components. If the interactions can be specified formally, it may be possible for synthesis systems to better manage the details of many interactions and constraints.

While correctness is not ensured solely by the construction of correctness proofs, such proofs can go a long way in convincing us that the software will behave as intended. Proving properties about arbitrary programs is difficult. It may be easier to prove properties are hold if we control the construction of the system rather than take arbitrary programs and attempt to prove properties (Fischer 2001).

Conclusion

In order to reduce the cost of building synthesis systems, the following must be achieved.

- We must be able to reuse knowledge and theories.
- We must be able to reuse synthesis tools and techniques.
- We must have a workforce familiar with techniques and their application.

Although several systems under development have attempted to address the first two issues, it is still difficult to port a knowledge base from one application to another. Many interesting and useful techniques have been developed, but incorporation of one technique into a different system is very difficult. Just as component libraries have facilitated the construction of object-oriented systems, we must construct synthesis components that can be matched and tailored to developing systems.

Finally, few computer science and software engineering professionals are trained to use formal techniques. The Software Engineering Body of Knowledge (SWEBOK 2001) composed by the IEEE does not have a chapter on formal methods. Most software engineering textbooks (see for example (Pfleeger 2001, Pressman 2000, Sommerville 1999)) make only passing mention of formal methods. Few software engineers are aware of the utility of synthesis techniques.

References

Dorner, D., 1996. *The Logic of Failure*, Cambridge, Mass.: Perseus Press.

Fischer, B., 2001. NASA Ames Research Center, personal communication.

Flener, P., 1995, *Logic Program Synthesis from Incomplete Information*, Norwell, Mass.: Klewer Academic Publishers.

Manna, Z., and Waldinger, R., 1980. A Deductive Approach to Program Synthesis, *ACM Transactions on Programming Languages and Systems*, 2(1), 90-121.

Pfleeger, S., 2001. *Software Engineering Theory and Practice*, Upper Saddle River, N.J.: Prentice-Hall.

Pressman, R., 2000. *Software Engineering: A Practitioner's Approach*, 4th Edition, Boston, Mass.: McGraw Hill.

Roach, S., Lowry, M., and Pressburger, T., 1995. Animating Observation Geometries with Amphion. NASA Information Systems Newsletter, 3(35) 35-38.

Roach, S., and Van Baalen, J., 1996. Automatic Program Synthesis in Amphion, a Simulation Package for Evaluating Space Probe Missions. In Proceedings of the Wyoming Space Grant Symposium, University of Wyoming.

Roach, S., Van Baalen, J., and Lowry, M., 1997. Meta-Amphion: Scaling up High Assurance Deductive Program Synthesis. In Proceedings of the IEEE High Integrity Software Symposium, 81-93. Albuquerque, New Mexico.

Roach, S., 1997. TOPS: Theory Operationalization for Program Synthesis, PhD diss., Dept. of Computer Science, University of Wyoming.

Smith, D., 1991. KIDS: A Knowledge-Based Software Development System. In *Automating Software Design*, M. Lowry and R. McCartney (eds.), 483-514. MIT Press.

Sommerville, I., *Software Engineering*, 5th Edition, Reading, Mass.: Addison-Wesley.

Srinivas, Y., and McDonald, J., 1996. The Architecture of Specware, a Formal Software Development System, Technical Report, KES.U.96.7, Kestrel Institute, Palo Alto, Calif.

Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I., 1994. Deductive Composition of Astronomical Software from Subroutine Libraries. In Proceedings of the 12th Conference on Automated Deduction. Nancy, France.

SWEBOK 2001. The Software Engineering Body of Knowledge, Ironman Version.

Van Baalen, J, and Roach, S., 1998. Using Decision Procedures to Build Domain-Specific Deductive Synthesis Systems. In Proceedings of LOPSTR'98 Eighth International Workshop on Logic Program Synthesis and Transformation. Manchester, UK.